Protocols for Distributed Management

Rolf Stadler

School for Electrical Engineering
KTH Royal Institute of Technology

Nov. 28, 2011

Why Distributed Management Protocols?

Drivers over the last 10 years:

- New requirements for management technology:
  increase scalability, robustness, level of self-configuration;
  reduce execution cycles.
- New technologies to manage:
  data center networks, cloud infrastructure and services.
- New actors demanding new solutions:
  Google, Amazon, Microsoft, Apple.

Enablers:

- Advances in distributed computing:
  gossip protocols, algorithms for virtual topologies,
  understanding of protocols in dynamic environments
- Enablers of network programmability:
  Juniper, Cisco provide open interfaces
  OpenFlow, SDN allow for programmable control and
  management planes

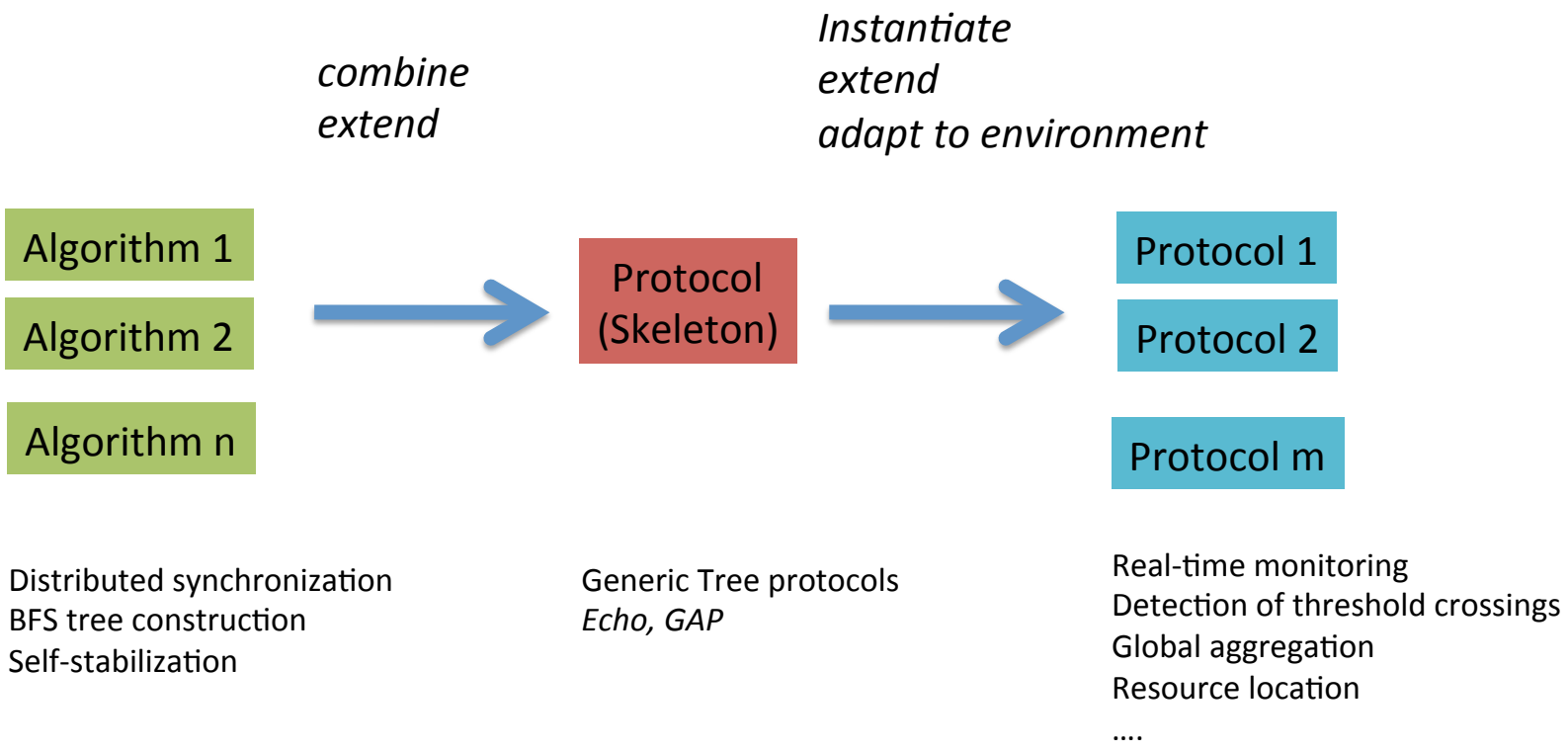
Goal of the Lecture

Objectives:

- Present two classes of tree-based management protocols.
- Show that properties of these protocols are inherited from fundamental algorithms underlying them.
- Discuss how these protocols are building blocks for more complex functions in the control and management planes.

Material:
A technical report covering the material of this lecture is available at:
www.ee.kth.se/~stadler/ProtocolsForDistributedManagement.pdf .

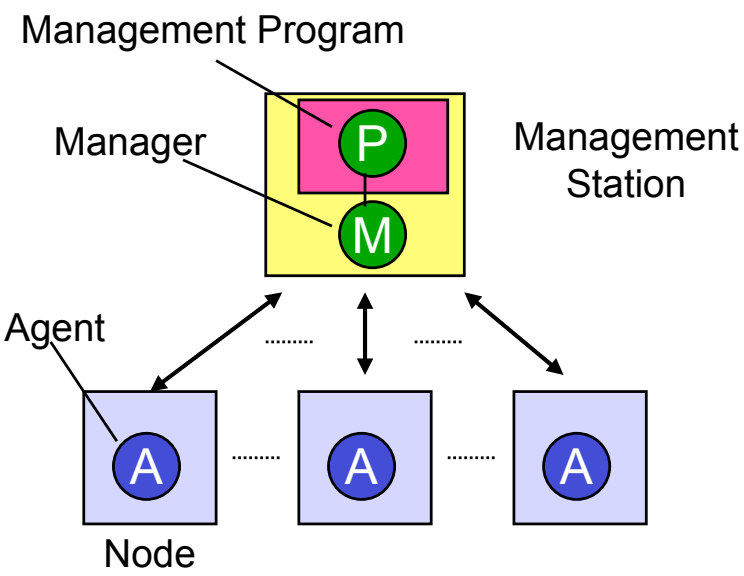
Methodology



Outline

- Approaches to Distributed Management
- An Architecture for Peer-to-peer Management
- The Echo Protocol for Distributed Management
- The GAP Protocol for Distributed Management

The Centralized Management Model



- Building block of traditional management systems, based on manager-agent interactions (polling or notification mode).
- Used in SNMP framework.
- Program P reads and writes MIB objects using a management protocol between the management station and network nodes.
- Program P cannot be executed on network nodes.

Assessment of the Centralized Management Model

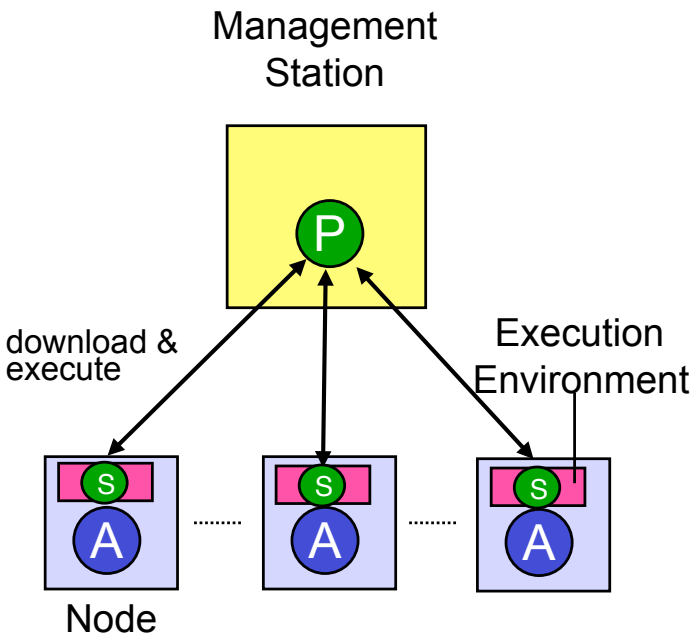
- Characteristics of the centralized model
  - Complexity of management operation is on the manager side; agents are "dumb".
  - Agents interact solely with manager, not among themselves.
- Drawbacks of the centralized model
  - Single point of failure—the management station can fail.
  - The centralized model is **not scalable**.
    Consider an operation that polls all nodes.
    - The management traffic generated (M: number of messages sent);
    - The load on the management station (L: number of request messages generated and response messages processed);
    - The execution time of a management operation (T: execution time of P);
    - M, L, T grow linearly with the number of managed nodes N, i.e., $M{=}O(N)$; $L{=}O(N)$; $T{=}O(N)$;
      for many management operations.
      Example: compute max link utilization.

Approaches to Distributed Management

- Hierarchies of (SNMP) Agents
- Script-enabled Agents

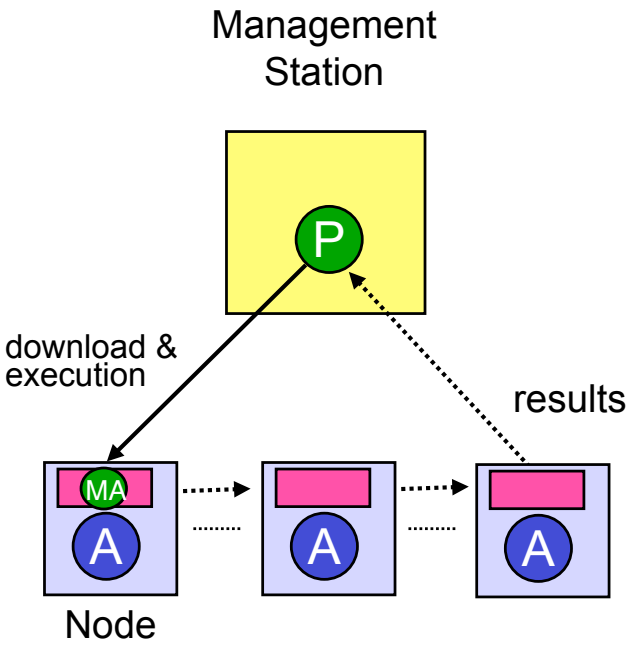
  Also called *Management by Delegation*
- Mobile Agents
- Peer-to-Peer Management

Script-enabled Agents



- A management program P downloads a script S onto one or more nodes.
- S runs in an execution environment (e.g., JVM) on the node. Results of this execution are sent back to P.
- S accesses A through a local interface,e.g., via SNMP.
- This approach is also known as *Management by Delegation*.
- Application scenarios: Statistical aggregation and notification schemes.

Mobile Agents (1)

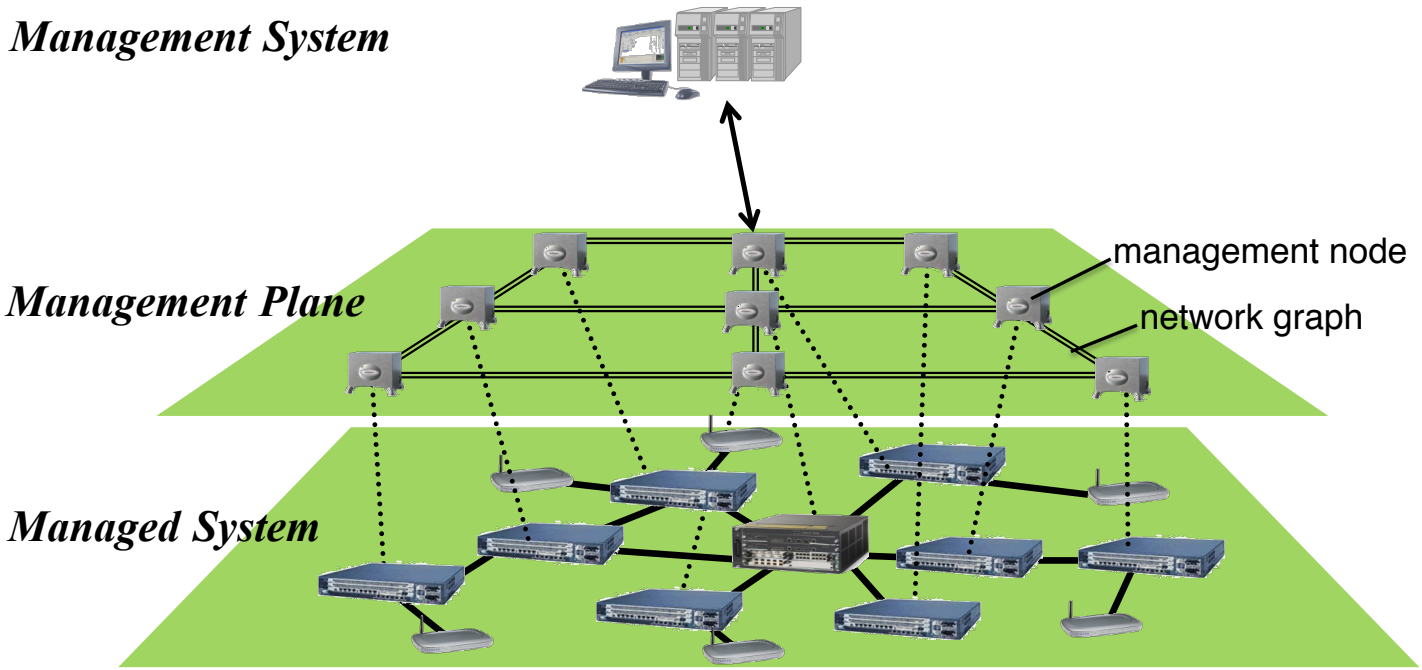


- A management program P downloads a mobile agent MA onto a node.
- MA is run on an execution environment on the node.
- MA accesses agent A through a local interface, e.g., via SNMP.
- MA can migrate to other nodes, taking its execution state with it.
- MA sends results of management operation back to P in form of messages.

Mobile Agents (2)

- A *mobile agent* is a self-contained program that can move from node to node in a network and that acts on behalf of a human operator or another entity.
- Proposed applications scenarios for management:
  - Diagnosing and correcting problems on nodes,
  - Updating software on nodes, etc.
- This approach is motivated by the potential to facilitate scalability and automated management.
- Drawbacks include security, safety, and the potential complexity of agents.
- To date, many research prototype, but few commercial applications of mobile agents in management.
- Many mobile agent platforms have been built 1995-2000.

An Architecture for Peer-to Peer Management



An architecture that supports the execution of distributed management protocols.

The Management Plane

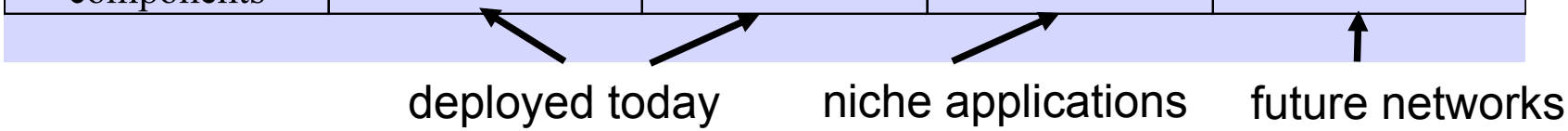
- The *management* plane conceptualizes the management resources inside the managed system.
- Each network node has an associated execution environment in the management plane, called *management node*, which represents processing, memory and storage capacity.
- A management node knows other nodes in its neighborhood and communicates with them through messages.
- This message network for peer interaction forms the *network graph*.
- *Management protocols* in this architecture are distributed algorithms that execute on this graph. They read and process state information in the management nodes and produce output that is available in one or more of them.

Realizing the Management Plane

- A management node can be realized as a virtual machine running
  (a) on a CPU inside a router,
  (b) on a blade that connects to a router backplane,
  (c) in an appliance that is situated close to a router.
- Depending on the specific realization of the management plane, the communication between a network element and its associated management node can take many forms, from using primitives for inter-thread communication to local SNMP interfaces.
- The network graph on the management plane can be realized as an overlay network.

Comparing Centralized Management with Distributed Management Approaches

	Centralized Management	Script-enabled Agents	Mobile Agents	Peer-to-peer Management
Main objective	design simplicity	decrease overhead	increase autonomy	increase parallelism
EE needed on managed nodes	no	yes	yes	yes
Code of operation is mobile	no	yes	yes	optional
State of operation is mobile	no	no	yes	yes
Complexity of mobile components	---	generally low	generally high	low

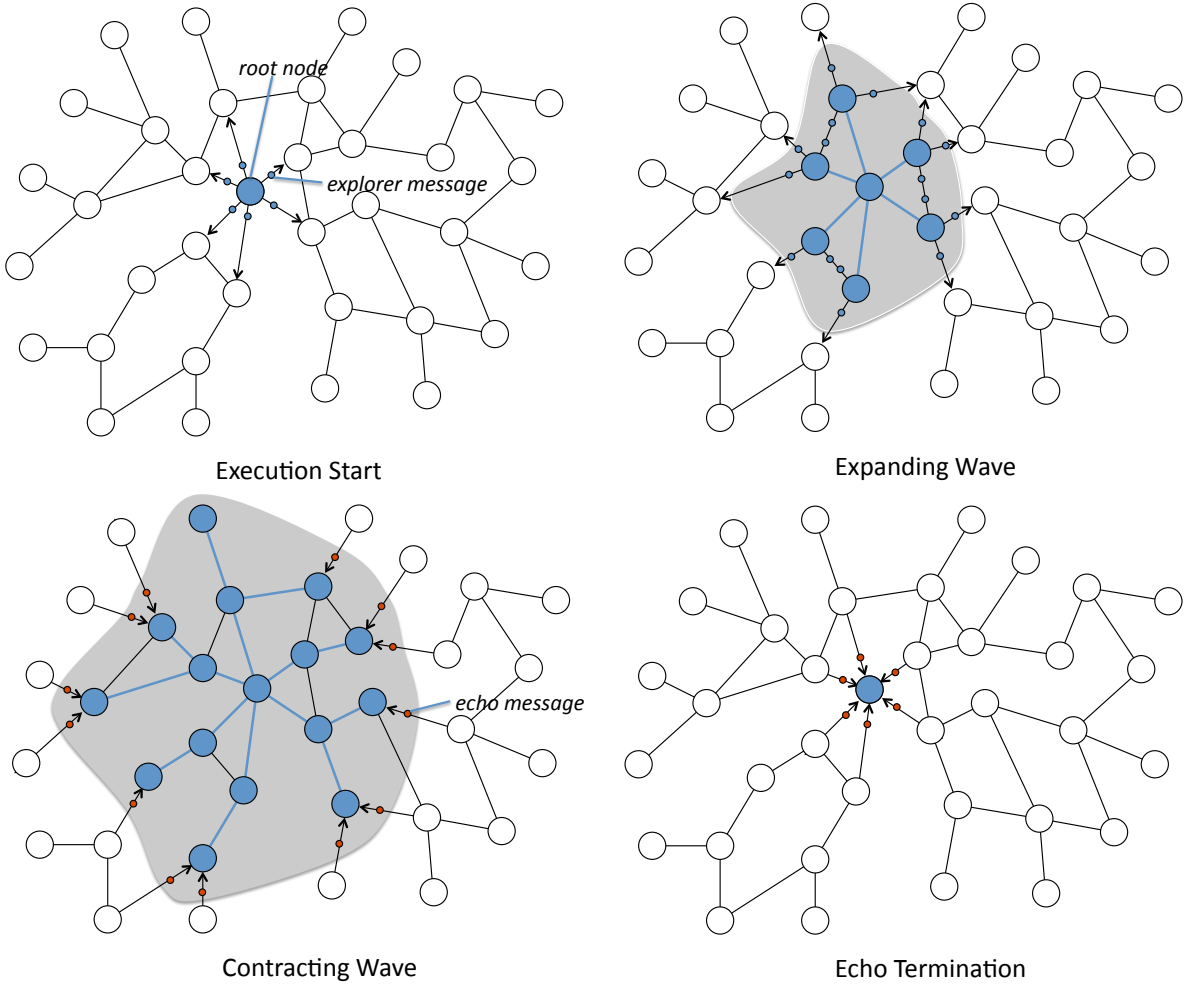


The Echo Protocol for Distributed Management

The echo protocol can be used for monitoring (distributed polling, global state estimation), resource discovery and distributed configuration.
Its execution can be sees as the expansion and contraction of a wave on the network graph.

- The execution starts and terminates on the *root* node.
- The wave expands through *explorer* messages, which nodes send to their neighbors.
- During expansion, local operations are triggered on the nodes after receiving an explorer.
- The results of these operations are collected in *echo* messages when the wave contracts.
- The aggregated result of the global operation becomes available at the root node.
- During expansion, echo constructs a spanning tree on the network graph, used for collecting and aggregating the partial results during contraction.

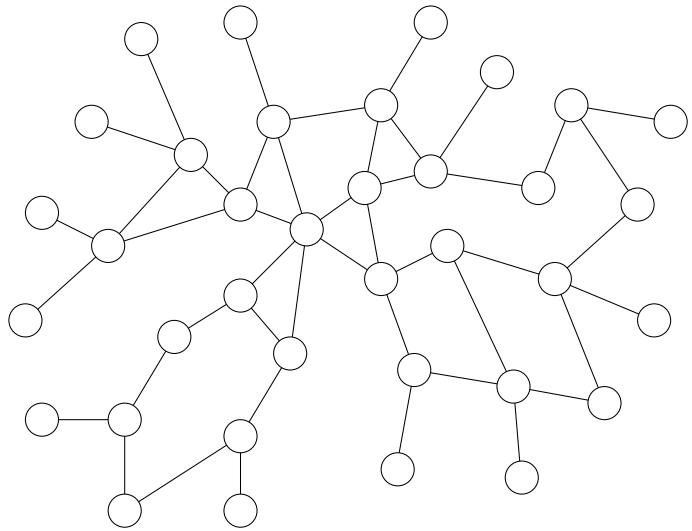
Execution of the Echo protocol



The echo algorithm executing on a network graph.

The Execution Model for Protocols on the Network Graph

- The network graph $G = (V, E)$ has bidirectional links and and is connected.
- Each node on G has a unique identifier and can distinguish its neighbors.
- Neighbors exchange messages $(TYPE, arg_1, ..., arg_n)$. They are read in the order received.
- We assume an asynchronous execution model with bounded delays.



The Echo algorithm by Segall (1)

message types:

1: (EXP, $from$)
2: (ECHO, $from$)

data structures:

3: N:= set of neighbors;

root node:

4: forall $n \in N$ send (EXP, root) to n;
5: **while** $N \neq \emptyset$ **do**
6: receive (ECHO, n);
7: $N := N - \{n\}$;
8: 'Echo completed';

non-root node v**:**

1: receive (EXP, n);
2: $parent := n$; $N := N - \{n\}$;
3: forall $n \in N$ send (EXP, v) to n;
4: **while** $N \neq \emptyset$ **do**
5: receive EXP or ECHO message from n;
6: $N := N - \{n\}$;
7: send (ECHO, n) to $parent$;

The Echo algorithm by Segall (2)

- The echo algorithm creates a spanning tree on the network graph.
- The initiating node is the root node of the tree.
- The $parent$ variable on each non-initiating node points to its parent on the spanning tree.
- During execution, each node sends an EXP message to each neighbor;
  it receives an EXP or ECHO message from each neighbor.
- The algorithm performs a distributed synchronization function; no numerical value is computed.
- The algorithm relies only on local information.

The Echo protocol

It extends the echo algorithm for network management.

- Each node of the network graph can be the root node of an execution.
- A local management operation is executed on each node during the expansion phase.
  The local results are aggregated during the contraction phase.
  The aggregate of all results is available at the root node at the end of the execution.
  These operations are defined in the local aggregator object.

Echo Message Types and Aggregator Object

message types:

1: (INVOKE, $invoker$) ▷ echo invoked by $invoker$
2: (RETURN, $result$) ▷ return result of echo operation
3: (EXP, $from$) ▷ explorer sent by $sender$
4: (ECHO, $from$, agg) ▷ echo with result agg sent by $sender$

aggregator object A:

5: A.initiate() ▷ initialize aggregate; perform local operation
6: A.aggregate() ▷ aggregate result from a child
7: A.global() ▷ perform operation on aggregate (root)
8: A.value() ▷ return the current value of (partial) aggregate

The aggregator object captures the semantics of the management operation.

Echo Protocol: Main Loop

```
1:  procedure ECHO( )
2:      N:= set of neighbors of node v;
3:      visited :=false;                          ▷ no EXP received
4:      while true do
5:          receive message;
6:          switch (message)
7:              case (INVOKE, invoker):           ▷ v is root
8:                  ...
9:              case (EXP, from):
10:                 ...
11:             case (ECHO, from, agg):
12:                 ...
13:         end switch
```

Pseudocode for node v.

Echo Protocol: message processing

```
1:  case (INVOKE, invoker):                                  ▷ v is root
2:      A.initiate();
3:      if N ≠ ∅ then                                        ▷ v is only node in G
4:          send (EXP, v) to nodes in N;
5:      else
6:          send (RETURN, A.global()) to invoker;

7:  case (ECHO, from, agg):
8:      A.aggregate(agg);
9:      N := N − {from};
10:     if N = ∅ then
11:         if v ≠ root then
12:             send (ECHO, A.value()) to parent;
13:         else
14:             send (RETURN, A.global()) to invoker;
```

Echo Protocol: message processing

```
1:  case (EXP, from):
2:      N := N − {from};
3:      if not visited then
4:          visited := true;
5:          parent := from;
6:          A.initiate();
7:          if N ≠ ∅ then
8:              send (EXP, v) to nodes in N;
9:          else
10:             send (ECHO, A.value()) to parent;          ▷ v is a leaf
11:     else
12:         do nothing;                  ▷ from is not neighbor of v on tree
```

Echo Aggregator: MaxLoad()

```
1:  aggregator object MaxLoad( )
2:   var: maxLoad: int;          ▷ the maximum link load locally known
3:       lmax: linkId;           ▷ link with maximum load locally known

4:   procedure initiate( )
5:    L := set of outgoing links;
6:    maxLoad := max_{link∈L} load(link);
7:    lmax := link in L with value maxLoad;
8:   procedure aggregate([childLoad: int; lchild: linkId])
9:    if childLoad > maxLoad then
10:     lmax := lchild;
11:     maxLoad := childLoad;
12:  procedure value( )
13:   return ([maxLoad, lmax])
14:  procedure global( )
15:   return ([maxLoad, lmax])   ▷ no function applied on aggregate

16:  function load(l: linkId)
17:   return current load on link l;
```

Echo Aggregator: AverageLoad()

```
1:  object AverageLoad()
2:    var: sumLoad := 0;    ▷ total load of the (sub)tree rooted v
3:         nLinks := 1;  ▷ number of network links of the (sub)tree

4:    procedure initiate( )
5:      L := set of outgoing links;
6:      sumLoad := Σ_{link∈L} load(link);
7:      nLinks := |L|;
8:    procedure aggregate([sumLoadChild: int; nLinksChild: int])
9:      sumLoad := sumLoad + sumLoadChild;
10:     nLinks := nLinks + nLinksChild;
11:   procedure value( )
12:     return ([sumLoad, nLinks])
13:   procedure global( )
14:     return (sumLoad/nLinks)

15:   function load(l: linkId)
16:     return current load on link l;
```

Echo-based Management operations(1)

Application: Distributed polling or estimation of a global aggregates.
Echo can compute a global function $F = F(x_1, ..., x_N)$ on local variables $x_i, i = 1, \ldots, N$, whereby each variable x_i is associated with a node of the network graph.
F can be computed in a single execution of echo:

- If F can be written using a binary function f that is both commutative $(f(x, y) = f(y, x) \forall x, y)$ and associative $(f(x, f(y, z)) = f(f(x, y), z) \forall x, y, z)$. Example: $sum()$.
- $average()$, which is not associative, can be computed as $average(x_1, ..., x_N) = sum(x_1, ..., x_N)/count(x_1, ..., x_N)$
- If F has form $F = (N(m_1), ..., N(m_k)$, whereby $m_1, ..., m_k$ are the possible values for x_i, $N(m_i)$ is the number of occurrences of m_i. Example: $histogram()$.

Assumption: time-scale of change of local variables is large compared with execution time of echo.

Echo-based Management operations (2)

Application: Network Search.
Echo traverses network graph during execution and performs local search on each node.
Example: *find set of routers that run IOS version x.y*.

Application: Perform operations on nodes with given properties.
Similar to network search, with different local operation.
Example: *update module z on all routers that run IOS version x.y*.

Performance of echo-based operations(1)

Performance metrics are obtained from properties of the echo algorithm, assuming bounds on communication delays between nodes and processing delays for local message processing.

- *Management traffic M* (message complexity): Protocol execution generates a balanced load on the network graph $G = (V, E)$, with 2 messages traversing each link in opposite direction. $M = 2 * |E|$ messages. The message sizes depends on the specific aggregation function.
- *Processing load L*: We measure load on a node as number of incoming messages. Load increases proportionally with the number of neighbors of a node, i.e., its degree. Or: $L = O(deg(G))$. ($deg(G)$ is the max degree of any node on G).
- *Execution time T* (time complexity): The execution time increases linearly with the height of the spanning tree, which is bounded by the diameter of the network graph, $diam(G)$. $T = O(diam(G))$ and $T = O(deg(G))$.

Performance of echo-based operations(2)

Comparing centralized management operation with echo-based operation:

For a network graph with $diam(G) = O(\log(N))$, echo exhibits
M =2 messages per link, $L \leq deg(G)$ messages per node,
$T = O(\log(N))$.
We call echo *scalable* on G with respect to the metrics M, L, T, since these metrics increase less than linear with the network size N.
For a management operation executed in the centralized model with polling, the management station experiences:
$M = 2N$ messages, $L = 2N$ messages, T is proportional to $2N$.

In small networks a centralized management operation can be more efficient than an equivalent echo-based operation.
In large networks (> 1000 nodes) an echo-based operation can significantly outperform a centralized one. The gain in scalability comes at the cost of a more complex management infrastructure.

Extensions for practical applications

The echo protocol must be extended and adapted for operational use. Examples:

- *Concurrent execution.* An invocation identifier allows for running several echo operations simultaneously in the management plane.
- *Restricted scope*. In a large network, the scope can be restricted, e.g., to n hops from root. (Consider that nodes with maximum hop count from the root may be involved more than once in the same execution.)
- *Stationary tree*. For echo-based periodic polling from the same root node, keep the spanning tree alive between runs. (The solution must maintain the tree in case the network graph changes.)
- *Robust echo*. The presented version of echo is not robust to certain changes to the network graph that result from node churn or failures. Possible approach to crash failures: introduce an event, triggered either by a timeout or a failure detector, that lets a waiting node resume protocol operation.

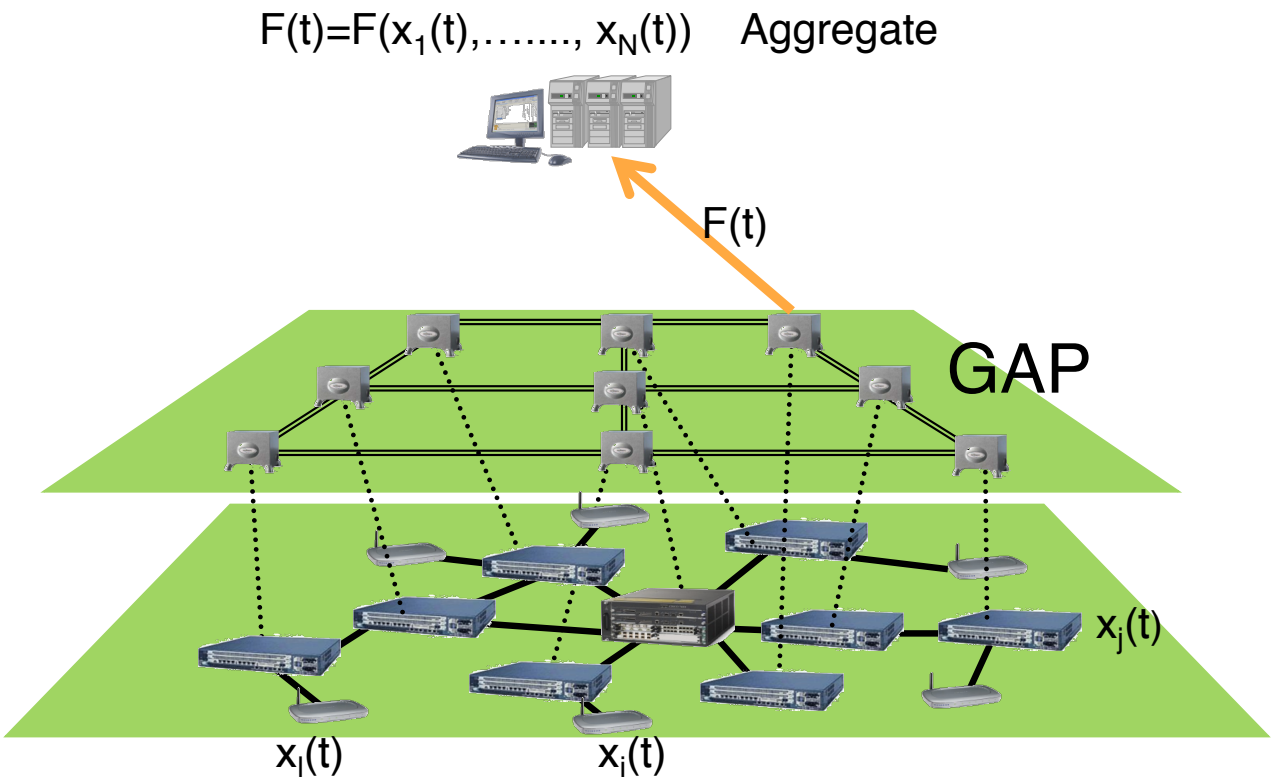
The GAP Protocol for Distributed Management

The GAP protocol (Generic Aggregation Protocol)

- executes on a bidirectional, connected network graph;
- provides a continuous estimate of a global aggregate, computed over local variables across all nodes;
- dynamically adapts to node churn and node failures;
- allows controlling the tradeoff between protocol overhead and estimation accuracy, by limiting the message rate on the network graph.

Its primary use is continuous real-time monitoring. It is a push protocol where updates to the local variables are 'pushed' upwards the tree towards the root. (Echo performs one-time estimations through polling.)
GAP creates a BFS (Breath-First Search) spanning tree on a connected network graph, which is used to perform incremental, distributed aggregation, with the result at the root.
Distributed algorithms that underlie GAP: Belman-Ford algorithm, DIM algorithm, tree-based aggregation algorithm.

Execution of the GAP protocol



The GAP protocol executes on the network graph and continuously computes a global function $F(t) = F(x_1(t), ..., x_N(t))$ on the local variables $x_i(t), i = 1, ..., N(t)$.

The Distributed Bellman-Ford Algorithm

messages:

1: (UPDATE, n, l); ▷ node n has distance l from root

root node:

2: $level := 0$; $parent$:=root;

3: send $(\text{UPDATE}, root, 0)$ to all neighbors on G;

non-root node v:

4: $level$:=infinite; $parent$:=undef;

5: **while** true **do**

6: read (UPDATE, n, l);

7: **if** $(level > l + 1)$ **then**

8: $level := l + 1$; $parent := n$;

9: send $(\text{UPDATE}, v, level)$ to all neighbors except $parent$;

The Distributed Bellman-Ford Algorithm (2)

The algorithm constructs a BFS (Breath-First Search) spanning tree on a connected network graph. A BFS tree connects each node to the root with a shortest path (e.g., minimal number of hops).

- Each node maintains a $level$ variable that indicates its distance to the root and a pointer to its parent node.
- The algorithm builds a spanning tree in a distributed fashion, starting from the root and continuing towards the leafs. The tree is encoded in the $parent$ variables.
- Nodes exchange messages (UPDATE, $n, level$), conveying that node n has (believed) distance $level$ from the root.
- The algorithm guarantees that the variables $level$ and $parent$ eventually contain correct values, once no more messages are exchanged.
- The algorithm has a time complexity of $O(diam(G))$ and a message complexity of $O(N * |E|)$ in an asynchronous model.
- Based on this algorithm, GAP builds up the spanning tree during initialization.

The DIM Algorithm

messages:

1: (UPDATE, n, l); ▷ node n has distance l from root

root node:

2: $level := 0$; $parent :=$root;

3: send $(\text{UPDATE}$, root, $0)$ to all neighbors on G;

non-root node v:

4: **while** true **do**

5: read (UPDATE, n, l);

6: $level_n := l$;

7: Among all neighbors of v with the smallest level, choose node with the smallest index k

8: $newlevel := level_k + 1$; $parent := k$;

9: **if** $(newlevel \neq level)$ **then**

10: send $(\text{UPDATE}, v, level)$ to all neighbors on G;

11: $level := newlevel$;

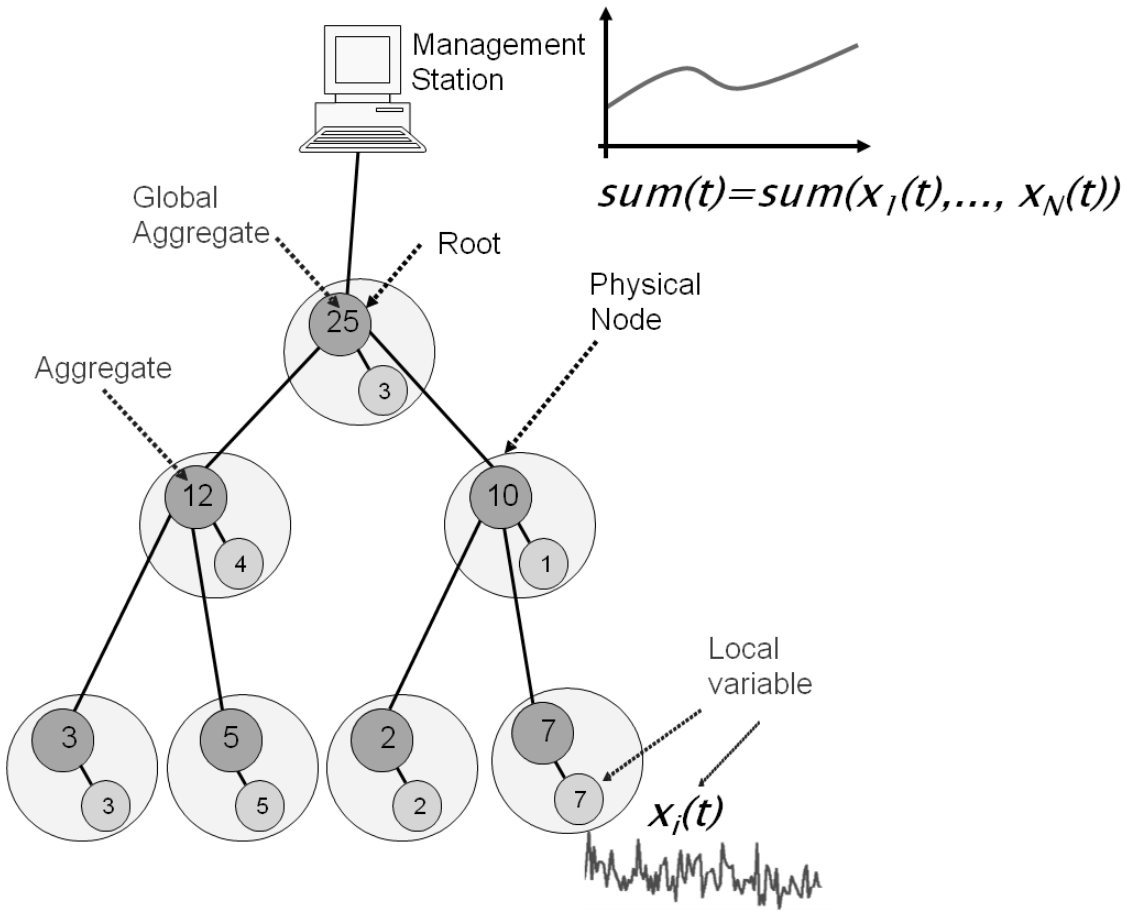
The DIM Algorithm (2)

The algorithm, developed by Dolev, Israeli and Moran, is *self-stabilizing*: If the root has the correct level, then, independent of initial values for $parent$ and $level \neq 0$, eventually, the $parent$ pointers form a BFS tree, and the $level$ variables contain the correct distance to the root.
DIM can be seen as a version of Belman-Ford where the the variables $parent$ and $level$ of non-root nodes are initialized with random values.
Other differences to Belman-Ford are:

- A non-root node maintains a variable $level_n$ for each neighbor n.
- A node identifies its neighbors through local indices. Among the neighbors with minimal level, the one with the smallest index is chosen as parent. (DIM and Belman-Ford may produce (slightly) different trees.)
- The UPDATE message is sent to all neighbors, including $parent$, as a node has knowledge about the level of all neighbors.

GAP makes use of DIM to create the BFS tree and maintain it in response to node churn and node failures.

The Algorithm for Incremental Aggregation on a Tree



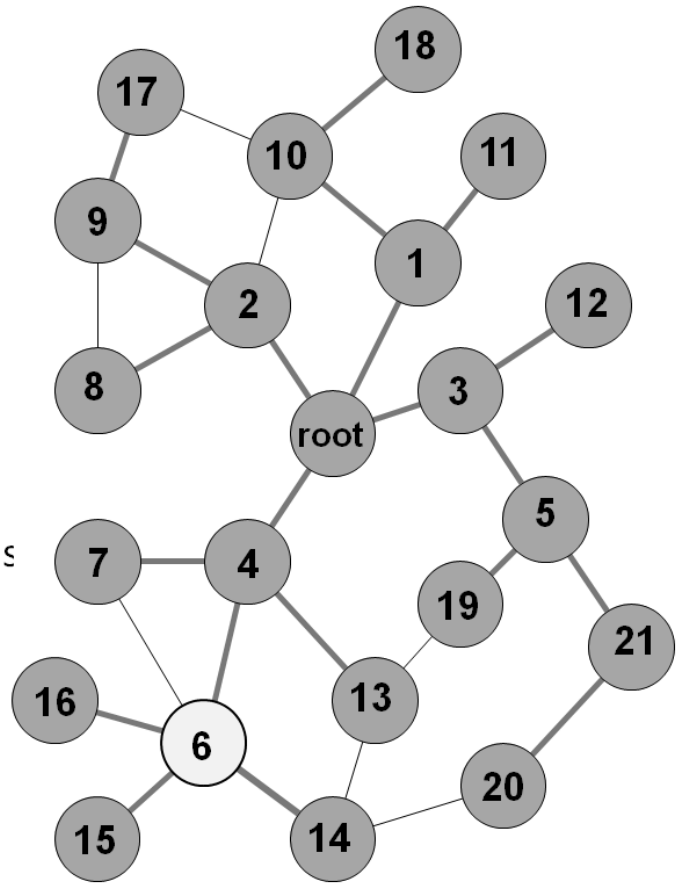
Incremental aggregation on a tree: the aggregate is computed bottom-up, from the leafs to the root. The aggregation function is sum in this example.

In GAP, the aggregation function is defined in the aggregator object.

The GAP Neighborhood Table

nodeId	status	level	aggregate
6	self	2	40
4	parent	1	70
7	peer	2	10
14	child	3	10
15	child	3	10
16	child	3	10

The numbers on the overlay show the node ids
The neighborhood table for node 6 is shown.
The update vector of the table is *(2, 4, 40).*
The aggregation function is sum.
Each node has a local variable with value 10.



GAP Neighborhood Table and Aggregator Object

GAP neighborhood table:

T: table with rows $(node, status, level, aggregate)$

table methods:

addEntry(n, s, l, a) ▷ add entry for node n

removeEntry(n) ▷ remove entry for node n

updateEntry(n, l, p, a) ▷ update entry for node n

updateVector$()$: (l, p, a) ▷ give $level, parent, aggregate$ of node

restoreTableInvariant$()$ ▷ maintain BFS tree

Aggregator object A

A.initiate() ▷ initiate update messages of change to local variable

A.aggregate() ▷ compute the (partial) aggregate of this node

A.global() ▷ perform an operation on the aggregate (root node)

The GAP Protocol, node v

messages:

1: (NEW, n) ▷ new neighbor n detected
2: (FAIL, n) ▷ neighbor n failed
3: $(\text{UPDATE}, n, l, p, a)$ ▷ node n has aggregate a, level l, parent p
4: $(\text{LOCALVAR}, x)$ ▷ the local variable has value x

```
 5: procedure GAP( )
 6:     T :=empty table;
 7:     if v =root then
 8:         addEntry(root, parent, -1, undef);
 9:         addEntry(root, self, 0, undef);
10:     else
11:         addEntry(v, self, undef, undef);
12:     vector :=updateVector();
13:     send (UPDATE, v, vector) to all neighbors;
14:     A.initiate();
15:     while true do
16:         ...
```

The GAP while loop

```
 1: while true do
 2:     read message;
 3:     switch (message)
 4:         case (NEW, from):
 5:             addEntry(from, peer, undef, undef);
 6:             send (UPDATE, v, vector) to from;
 7:         case (FAIL, from):
 8:             removeEntry(from);
 9:         case (LOCALVAR, x):
10:             empty;
11:         case (UPDATE, from, level, parent, aggregate):
12:             updateEntry(from, level, parent, aggregate);
13:     end switch
14:     restoreTableInvariant();
15:     A.aggregate(); if (v =root) then A.global();
16:     newvector :=updateVector();
17:     if newvector ≠ vector then
18:         send (UPDATE, v, newvector) to all neighbors;
19:         vector := newvector;
```

From the DIM Algorithm to the GAP protocol

Spanning tree construction and maintenance:

- *T*he DIM variables $level$, $level_n$ and $parent$ values are stored in the GAP neighborhood table.
- *T*he DIM computation of $level$ and $parent$ is contained in

  restoreTableInvariant()

  1: Among all neighbors of v with the smallest level,
  choose node with the smallest index k

  2: $level := level_k + 1$; $parent := k$;
- *N*ew message types NEW and FAIL capture node addition and failure.
  Adding a node to or removing it from the network graph "corrupts" the spanning tree, which is "re-built" using DIM's self-stabilizing property.
- *G*AP sends a message to neighbors only when local state changes.

Introducing incremental aggregation of local variables.

- *A*n aggregator object defines the aggregation function.
  The message LOCALVAR indicates a change in local variable.

Performance of the GAP protocol (1)

Performance metrics are obtained from properties of the GAP algorithm, assuming bounds on communication delays between nodes and processing delays for local message processing.

- *Management traffic M and processing load L*: For GAP with rate control, M is limited to r messages per sec for each link on the network graph G. The maximum possible processing load on a node, measured in incoming messages per sec, $L = O(r * deg(G))$. r can be used to control both M and L. Both metrics become independent of the network size (for graphs with bounded degree), which makes GAP suitable for large networked systems.
- *Time for initialization*: The time from starting the protocol on all nodes to the root having the correct aggregate is proportional to the height h of the aggregation tree and thus proportional to the diameter $diam(G)$ of the network graph (if the local values do not change during initialization).
  h rounds are needed for the node with the longest distance from the root do have the correct level information; an additional h rounds for an update from that node to reach the root.

Performance of the GAP protocol (2)

- *Time for update of aggregate and reconfiguration due to node churn or failure*: An update to a local variable triggers an update of the aggregate on the root within h rounds. Adding a node or removing a node takes at most $2 * diam(G)$ rounds, until the spanning tree has adapted to the new topology of G.
- *Dependence on the on network graph:* Most performance metrics depend on the topology of G, in a similar way as the performance of echo does. The traffic and processing loads, initialization, update and reconfiguration times, all depend on the topology of the spanning tree, which, in turn depends on G.
  For a network graph with $diam(G) = O(log(N))$, the initialization time, the update time and the reconfiguration time of GAP are all $O(log(N) * deg(G))$.

Extensions for operational use (1)

While the GAP pseudocode contains a complete protocol, it should be regarded as a skeleton for a practical implementation.

- *Invocation parameters:* Some examples: parameters that identify the root node, the specific aggregator, the local variable(s) to be aggregated, the scope, the maximum message rate r for communication between neighboring nodes, etc. Also, the protocol must be extended to enable multiple concurrent invocations and to include a mechanism for terminating an invocation.
- *Robustness to node churn and crash failures:* Since GAP is self-stabilizing, it is robust to node churn and crash failures, if the network graph stays connected. The root node can neither leave nor fail. Further attention is needed:
  - during the transition phase when the tree reconstructs, significant errors in estimating the aggregate can occur;
  - root failures must be handled;
  - the case of partitioning of the network graph must be handled. GAP runs correctly on the subgraph that contains the root, not on the other subgraphs.

Extensions for operational use (2)

- *Synchronized aggregation.* GAP accurately estimates aggregates assuming that computational and communication delays can be neglected. This assumption holds for many scenarios. Otherwise, to avoid errors, GAP must be be extended. (See report.)
- *Distance metrics other than hop count:* GAP keeps executing correctly when the distance metric, which determines a node's $level$, is changed, for instance to link delay. This is due to the properties of the Bellman-Ford algorithm.
- *Global aggregate available on all nodes:* GAP computes the global aggregate at the root node only; One can extend the aggregation mechanism in an elegant way so that the global aggregate becomes available on all nodes. (See assignments.)

Assignment 1: Echo Aggregators

Assume that each router in a network domain maintains a table that lists the current IP flows through the router in the following form:
(flowId, protocol, bandwidth, next hop).

Write two echo aggregators, as follows:
(a) MaxFlowBw() identifies the flow with the highest bandwidth in the network and computes the path of this flow through the network.
(b) BwPercentage() computes the percentage of bandwidth currently used by various protocols. (Compute this by summing up, for each protocol, the bandwidth over all flows.)
(c) Determine the size of the echo messages from these two aggregators.

Assignment 2: Robust Echo

Write a version of the echo protocol that is robust to node (crash) failures.
During the execution of robust echo, one or more non-root nodes may fail. After execution, robust echo sends the result of the operation to the invoking node, together with a flag that indicates whether a failure occurred. Use the same approach as the GAP protocol, by introducing a message type (FAIL, n) whereby a failure detector sends a message to all neighbors of node n after node n fails.

Assignment 3: GAP with global aggregate on all nodes

Modify the GAP protocol so that each node locally computes the global aggregate.
A naive approach to this problem is one where the root broadcasts updates to the global aggregate, using the spanning tree.
You should follow a second approach, based on the idea that each node v on the spanning tree can be seen as the root of an aggregation tree with the same topology as the spanning tree. To compute the global aggregate at v, apply the same mechanism that GAP uses to compute the global aggregate at the root node of the spanning tree. For each neighbor n on the tree, v computes a separate value for the aggregate in the update vector sent to n. This value is computed as the aggregate over the local variable of v and the aggregates of all its neighbors except n.
Compare the naive solution and your solution with respect to the communication overhead and the update time for an update of a single local variable.

References (1)

The Segall echo algorithm is discussed in:


A. Segall.
Distributed network protocols.
IEEE Transactions on Information Theory, 29:23–35, 1983.

Echo and related algorithms are discussed in:


G. Tel.
An Introduction to Distributed Algorithms.
Cambridge University Press, second edition edition, 2000.

The distributed Belman-Ford algorithm is discussed in:


D. Peleg.
Distributed Computing. A Locality-Sensitive Approach.
SIAM Monographs on Discrete Mathematics and Applications, 2000.

The DIM algorithm is discussed in:


S. Dolev.
Self-Stablilization.
MIT Press, 2000.

References (2)

An extension of echo for flow monitoring using an SQL interface:

K.S. Lim and R. Stadler.

Real-time views of network traffic using decentralized management.
In *9th IFIP/IEEE International Symposium on Integrated Network Management (IM 2005)*, 2005.

The GAP protocol is introduced in:

M. Dam and R. Stadler.

A generic protocol for network state aggregation.
In *RVK 05, Linkping, Sweden, June 14-16*, 2005.

Various extensions of the GAP protocol:

A. Gonzalez Prieto and R. Stadler.

A-gap: An adaptive protocol for continuous network monitoring with accuracy objectives.
Network and Service Management, IEEE Transactions on, 4(1):2–12, June 2007.

F. Wuhib, M. Dam, and R. Stadler.

Decentralized detection of global threshold crossings using aggregation trees.
Computer Networks, 52(9):1745–1761, February 2008.

D. Jurca and R. Stadler.

H-gap: Estimating histograms of local variables with accuracy objectives for distributed real-time monitoring.
Network and Service Management, IEEE Transactions on, 7(2), June 2010.